Community Experience Distilled

Spring Essentials

Build mission-critical enterprise applications using Spring Framework and Aspect Oriented Programming

Shameer Kunjumohamed
Hamidreza Sattari

[PACKT] PUBLISHING open source* community experience distilled

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 4 **'Understanding WebSocket'**
- A synopsis of the book's content
- More information on **Spring Essentials**

About the Authors

Shameer Kunjumohamed is a software architect specializing in Java-based enterprise application integrations, SOA, and the cloud. Besides Java, he is well-versed in the Node.js and Microsoft .NET platforms. He is interested in JavaScript MVC frameworks such as EmberJS, AngularJS, and ReactJS.

Shameer has co-authored another book, *Spring Web Services 2 Cookbook*, *Packt Publishing* with Hamidreza Sattari, who is the co-author of this book as well.

Based in Dubai, UAE, Shameer has over 15 years of experience in various functional domains. He currently works as a principal applications architect for a major shipping company in Dubai.

Hamidreza Sattari is an IT professional and has worked in several areas of software engineering, from programming to architecture as well as management. He holds a master's degree in software engineering from Herriot Watt University, UK. In recent years, his areas of interest have been software architecture, data science, and machine learning. He co-authored the book *Spring Web Services 2 Cookbook*, *Packt Publishing* in 2012. He maintains the blog `http://justdeveloped-blog.blogspot.com/`.

Preface

There are a lot of books written about Spring Framework and its subprojects. A multitude of online references are also available. Most of these massive resources discuss Spring in a lot of detail, which makes learning Spring a very time-consuming and sometimes tedious effort. The idea of this book is to allow novice Java developers or architects to master Spring without spending much time and effort and at the same time provide them with a strong foundation on the topic in order to enable them to design high-performance systems that are scalable and easily maintainable.

We have been using Spring Framework and its subprojects for more than a decade to develop enterprise applications in various domains. While the usage of Spring quickly raises the design quality of projects with its smart templates and subframeworks abstracting many error-prone and routine programming tasks, a developer needs a thorough understanding of its concepts, features, best practices, and above all, the Spring programming model in order to utilize Spring to its best.

We have seen Spring used wrongly inside many projects mainly because the developer either didn't understand the right use of a particular Spring component or didn't bother to follow the design approach Spring suggests for that component. Often, developers didn't appear to have the right knowledge of Spring Framework; when asked, their complaint mostly was the uphill task of learning such a vast framework from huge documentation. Most of this category of developers find Spring a mammoth framework that is difficult to learn, which is not really true.

Spring, if the basics are understood correctly, is very easy to conquer further. A developer needs to understand the Spring style of programming and architecting applications, and the result will be a piece of art. The design will look simple, pretty straightforward, and easily understandable, which is very important for the evolution of applications in the long run. This book is an attempt to fill that gap and provide a very solid foundation in Spring, its concepts, design styles, and best practices, in a very quick and easy way.

This book tries to engage the reader by providing the feeling of developing a realistic, modern enterprise application using Spring and its necessary features while giving him or her a solid understanding of its concepts, benefits, and usage with real-life examples. It covers the most important concepts and features of Spring Framework and a few of its critical subprojects that are necessary for building modern web applications.

The goal of Spring is to simplify enterprise application development. We hope this book simplifies mastering Spring so that developers can build smarter systems that make the world a better place.

What this book covers

Chapter 1, *Getting Started with Spring Core*, introduces the core Spring Framework, including its core concepts, such as POJO-based programming, Dependency Injection, and Aspect Oriented Programming, to the reader. It further explains the Spring IoC container, bean configurations, Spring Expression Language (SpEL), resource management, and bean definition profiles, all which become the foundation for the advanced topics.

Chapter 2, *Building the Web Layer with Spring Web MVC*, gives in-depth coverage of the Spring MVC web framework with its features and various different ways of configuring and tuning web applications using Spring. The chapter covers the building of both view-based web applications and REST APIs with many available options, including asynchronous request processing.

Chapter 3, *Accessing Data with Spring*, discusses the different data-access and persistence mechanisms that Spring offers, including the Spring Data family of projects, such as Spring Data JPA and Spring Data Mongo. This chapter enables the reader to design an elegant data layer for his or her Spring application, delegating all the heavylifting to Spring.

Chapter 4, *Understanding WebSocket*, discusses the WebSocket technology, which is gaining wider usage inside modern web applications, where low latency and high frequency of communication are critical. This chapter explains how to create a WebSocket application and broadcast a message to all subscribed clients as well as send a message to a specific client, and shows how a broker-based messaging system works with STOMP over WebSocket. It also shows how Spring's WebSocket fallback option can tackle browser incompatibility.

Chapter 5, *Securing Your Applications*, teaches the reader how to secure his or her Spring applications. It starts with authentication and explains Spring flexibility on authorization. You learn how to integrate your existing authentication framework with Spring. On authorization, it shows how to use Spring EL expressions for web, method, and domain object authorization. It also explains the OAuth 2.0 Authorization Framework and how to allow third-party limited access to user's protected resources on a server without sharing user's username and password.

Chapter 6, *Building a Single-Page Spring Application*, demonstrates how Spring can be used as the API server for modern single-page applications (SPAs) with an example of an Ember JS application. At first, it introduces the concept of SPAs, and then it explores using Ember JS to build the SPA. Finally, it covers building the backend API that processes requests asynchronously using Spring MVC and implements persistence using Spring Data JPA.

Chapter 7, *Integrating with Other Web Frameworks*, demonstrates how Spring can be integrated with Java web frameworks such as JSF and Struts so that even web applications not based on Spring MVC can leverage the power of Spring.

4
Understanding WebSocket

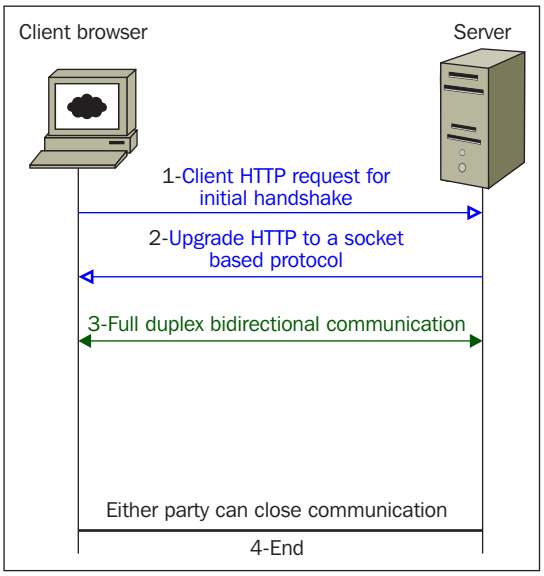
The idea of web applications was built upon a simple paradigm. In a unidirectional interaction, a web client sent a request to a server, the server replied to the request, and the client rendered the server's response. The communication started with a client-side request and ended with the server's response.

We built our web applications based on this paradigm; however, some drawbacks existed in the technology: the client had to wait for the server's response and refresh the browser to render it. This unidirectional nature of the communication required the client to initiate a request. Later technologies such as AJAX and long polling brought major advantages to our web applications. In AJAX, the client initiated a request but did not wait for the server's response. In an asynchronous manner, the AJAX client-side callback method got the data from the server and the browsers' new DHTML features rendered the data without refreshing the browser.

Apart from unidirectional behavior, the HTTP dependencies of these technologies required the exchange of extra data in the form of HTTPS headers and cookies. This extra data caused latency and became a bottleneck for highly responsive web applications.

WebSocket reduced kilobytes of transmitted data to a few bytes and reduced latency from 150 milliseconds to 50 milliseconds (for a message packet plus the TCP round trip to establish the connection), and these two factors attracted the Google's attention (Ian Hickson).

WebSocket (RFC 6455) is a full duplex and bidirectional protocol that transmits data in the form of frames between client and server. A WebSocket communication, as shown in the following figure, starts with an HTTP connection for a handshake process between a client and a server. Since firewalls let certain ports be open to communicate with the outside, we cannot start with the WebSocket protocol:



WebSocket communication

During the handshake process, the parties (client and server) decide which socket-based protocol to choose for transmitting data. At this stage, the server can validate the user using HTTP cookies and reject the connection if authentication or authorization fails.

Then, both parties upgrade from HTTP to a socket-based protocol. From this point onward, the server and client communicate on a full duplex and bidirectional channel on a TCP connection.

Either the client or server can send messages by streaming them into frame format. WebSocket uses the heartbeat mechanism using ping/pong message frames to keep the connection alive. This looks like sending a ping message from one party and expecting a pong from the other side. Either party can also close the channel and terminate the communication, as shown in the preceding diagram.

Like a web URI relies on HTTP or HTTPS, WebSocket URI uses `ws` or `wss` schemes (for example, `ws://www.sample.org/` or `wss://www.sample.org/`) to communicate. WebSocket's `ws` works in a similar way to HTTP by transmitting non-encrypted data over TCP/IP. By contrast, `wss` relies on **Transport Layer Security** (**TLS**) over TCP, and this combination brings data security and integrity.

A good question is where to use WebSocket. The best answer is to use it where low latency and high frequency of communication are critical—for example, if your endpoint data changes within 100 milliseconds and you expect to take very quick measures over the data changes.

Spring Framework 4 includes a new Spring WebSocket module with Java WebSocket API standard (JSR-356) compatibility as well as some additional value-adding features.

While using WebSocket brings advantages to a web application, a lack of compatibility in a version of some browser blocks WebSocket communication. To address this issue, Spring 4 includes a fallback option that simulates the WebSocket API in case of browser incompatibility.

WebSocket transmits data in the frame format, and apart from a single bit to distinguish between text and binary data, it is neutral to the message's content. In order to handle the message's format, the message needs some extra metadata, and the client and server should agree on an application-layer protocol, known as a **subprotocol**. The parties choose the subprotocol during the initial handshake.

WebSocket does not mandate the usage of subprotocols, but in the case of their absence, both the client and server need to transmit data in a predefined style standard, framework-specific, or customized format.

Spring supports **Simple Text Orientated Messaging Protocol** (**STOMP**) as a subprotocol—known as STOMP over WebSocket—in a WebSocket communication. Spring's Messaging is built upon integration concepts such as messaging and channel and handler, along with annotation of message mapping. Using STOMP over WebSocket gives message-based features to a Spring WebSocket application.

Using all of these new Spring 4 features, you can create a WebSocket application and broadcast a message to all subscribed clients as well as send a message to a specific user. In this chapter, we start by creating a simple Spring web application, which will show how to set up a WebSocket application and how a client can send and receive messages to or from an endpoint. In the second application, we will see how Spring WebSocket's fallback option can tackle browser incompatibly, how a broker based messaging system works with STOMP over WebSocket, and how subscribed clients can send and receive messages. In the last web application, however, we will show how we can send broker-based messages to a specific user.

Creating a simple WebSocket application

In this section, while developing a simple WebSocket application, we will learn about WebSocket's client and server components. As mentioned earlier, using a subprotocol is optional in a WebSocket communication. In this application, we have not used a subprotocol.

First of all, you need to set up a Spring web application. In order to dispatch a request to your service (called a handler in Spring WebSocket), you need to set up a framework Servlet (dispatcher Servlet). This means that you should register `DispatcherServlet` in `web.xml` and define your beans and service in the application context.

Setting up a Spring application requires you to configure it in XML format. Spring introduced the Spring Boot module to get rid of XML configuration files in Spring applications. Spring Boot aims at configuring a Spring application by adding a few lines of annotation to the classes and tagging them as Spring artifacts (bean, services, configurations, and so on). By default, it also adds dependencies based on what it finds in the classpath. For example, if you have a web dependency, then Spring Boot can configure Spring MVC by default. It also lets you override this default behavior. Covering Spring Boot in complete detail would require a full book; we will just use it here to ease the configuration of a Spring application.

These are the Maven dependencies of this project:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.5.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-websocket</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-messaging</artifactId>
    </dependency>
            <dependency>
        <groupId>org.json</groupId>
        <artifactId>json</artifactId>
        <version>20140107</version>
    </dependency>
</dependencies>
```

```
<properties>
    <java.version>1.8</java.version>
</properties>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

As mentioned in the beginning of this section, there is no subprotocol (and, subsequently, no application-layer framework) to interpret WebSocket messages. This means that the client and server need to handle the job and be aware of the message's format.

On the server's side, the handler (endpoint) receives and extracts the message and replies back (based on the business logic) to the client. In Spring, you can create a customized handler by extending either `TextWebSocketHandler` or `BinaryWebSocketHandler`. `TextWebSocketHandler` handles string or text messages (such as JSON data) and `BinaryWebSocketHandler` handles binary messages (such as image or media data). Here is a code listing that uses `TextWebSocketHandler`:

```
public class SampleTextWebSocketHandler extends
TextWebSocketHandler {
   @Override
    protected void handleTextMessage(WebSocketSession session,
    TextMessage message) throws Exception {
        String payload = message.getPayload();
        JSONObject jsonObject = new JSONObject(payload);
        StringBuilder builder=new StringBuilder();
        builder.append("From Myserver-").append("Your
        Message:").append(jsonObject.get("clientMessage"));
        session.sendMessage(new TextMessage(builder.toString()));
   }
}
```

Since we process only JSON data here, the class `SampleTextWebSocketHandler` extends `TextWebSocketHandler`. The method `handleTextMessage` obtains the client's message by receiving its payload and converting it into JSON data, and then it sends a message back to the client.

In order to tell Spring to forward client requests to the endpoint (or handler here), we need to register the handler:

```
@Configuration
@EnableWebSocket
public class SampleEhoWebSocketConfigurer {
    @Bean
    WebSocketConfigurer webSocketConfigurer(final WebSocketHandler
    webSocketHandler) {
        return new WebSocketConfigurer() {
            @Override
            public void registerWebSocketHandlers
            (WebSocketHandlerRegistry registry) {
                registry.addHandler(new
                SampleTextWebSocketHandler(), "/path/wsAddress");
            }
        };
    }
    @Bean
    WebSocketHandler myWebsocketHandler() {
        return new SampleTextWebSocketHandler();
    }
```

`@Configuration` and `@EnableWebsocket` tell Spring this is the WebSocket configurator of the project. It registers our handler (`SampleTextWebSocketHandler`) and sets the request path (in a WebSocket URL, such as `ws://server-ip:port/path/wsAddress`) that will be forwarded to this handler.

And now the question is how to set up a Spring application and glue all of this stuff together. Spring Boot provides an easy way to set up a Spring-based application with a configurable embedded web server that you can "just run":

```
package com.springessentialsbook.chapter4;
...
@SpringBootApplication
public class EchoWebSocketBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(EchoWebSocketBootApplication
        .class, args);
    }
}
```

`@SpringBootApplication` tags the `EchoWebSocketBootApplication` class as a special configuration class of your application and `@SpringBootApplication` behaves like the following annotations:

- `@Configuration`, which declares the class as a bean definition of an application context
- `@EnableAutoConfiguration`, which lets Spring Boot add a dependent bean definition based on the classpath (for example, `spring-webmvc` in the project classpath tells Spring Boot to set up a web application with its `DispatcherServlet` registration in `web.xml`)
- `@ComponentScan`, which is used to scan all annotations (services, controllers, configurations, and so on) within the same package (`com.springessentialsbook.chapter4`) and configure them accordingly

Finally, the `main` method calls `SpringApplication.run` to set up a Spring application within a web application without writing a single line of XML configuration (`applicationContext.xml` or `web.xml`).

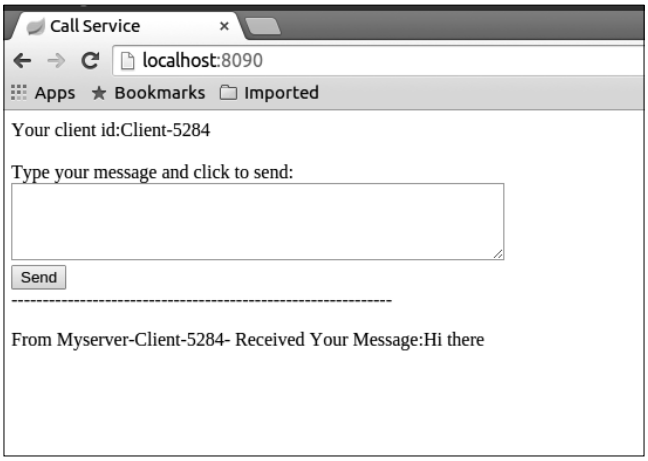
When a client wants to send a WebSocket request, it should create a JavaScript client object (`ws = new WebSocket('ws://localhost:8090/path/wsAddress')`) and pass the WebSocket service address. In order to receive the data, we need to attach a callback listener (`ws.onmessage`) and an error handler (`ws.onerror`), like so:

```
function openWebSocket(){
    ws = new WebSocket(
    'ws://localhost:8090/path/wsAddress');
    ws.onmessage = function(event){
        renderServerReturnedData(event.data);
    };

    ws.onerror = function(event){
        $('#errDiv').html(event);
    };
}

function sendMyClientMessage() {
    var myText = document.getElementById('myText').value;
    var message=JSON.stringify({ 'clientName': 'Client-
    '+randomnumber, 'clientMessage':myText});
    ws.send(message);
    document.getElementById('myText').value='';
}
```

You can run the application by running this command:

```
mvn spring-boot:run -Dserver.port=8090
```

This runs and deploys the web application on an embedded server on port `8090` (`8080` is not used here as it may conflict with your running Apache service). So, the index page of the application will be accessible at `http://localhost:8090/` (follow the instructions in `read-me.txt` to run the application). It should look like this:



The opening page of the application in a Chrome browser

When a user sends a text in Chrome, it will be handled by `SampleTextWebSocketHandler`, the handler will reply, and the response will be rendered in the browser.

If you try to test this application in a version of Internet Explorer lower than 10, you will get a JavaScript error.

As we discussed earlier, certain versions of browsers do not support WebSocket. Spring 4 provides a fallback option to manage these types of browsers. In the next section, this feature of Spring will be explained.

STOMP over WebSocket and the fallback option in Spring 4

In the previous section, we saw that in a WebSocket application that does not use subprotocols, the client and server should be aware of the message format (JSON in this case) in order to handle it. In this section, we use STOMP as a subprotocol in a WebSocket application (this is known as **STOMP over WebSocket**) and show how this application layer protocol helps us handle messages.

The messaging architecture in the previous application was an asynchronous client/server-based communication.

The `spring-messaging` module brings features of asynchronous messaging systems to Spring Framework. It is based on some concepts inherited from Spring Integration, such as messages, message handlers (classes that handle messages), and message channels (data channels between senders and receivers that provide loose coupling during communication).

At the end of this section, we will explain how our Spring WebSocket application integrates with the Spring messaging system and works in a similar way to legacy messaging systems such as JMS.

In the first application, we saw that in certain types of browsers, WebSocket communication failed because of browser incompatibility. In this section, we will explain how Spring's fallback option addresses this problem.

Suppose you are asked to develop a browser-based chat room application in which anonymous users can join a chat room and any text sent by a user should be sent to all active users. This means that we need a topic that all users should be subscribed to and messages sent by any user should be broadcasted to all. Spring WebSocket features meet these requirements. In Spring, using STOMP over WebSocket, users can exchange messages in a similar way to JMS. In this section, we will develop a chat room application and explain some of Spring WebSocket's features.

The first task is to configure Spring to handle STOMP messages over WebSocket. Using Spring 4, you can instantly configure a very simple, lightweight (memory-based) message broker, set up subscription, and let controller methods serve client messages. The code for the `ChatroomWebSocketMessageBrokerConfigurer` class is:

```
package com.springessentialsbook.chapter4;
…..
@Configuration
@EnableWebSocketMessageBroker
public class ChatroomWebSocketMessageBrokerConfigurer extends
AbstractWebSocketMessageBrokerConfigurer {
   @Override
   public void configureMessageBroker(MessageBrokerRegistry
   config) {
      config.enableSimpleBroker("/chatroomTopic");
      config.setApplicationDestinationPrefixes("/myApp");
   }
   @Override
   public void registerStompEndpoints(StompEndpointRegistry
   registry) {
      registry.addEndpoint("/broadcastMyMessage").withSockJS();
   }
}
```

`@Configuration` tags a `ChatroomWebSocketMessageBrokerConfigurer` class as a Spring configuration class. `@EnableWebSocketMessageBroker` provides WebSocket messaging features backed by a message broker.

The overridden method `configureMessageBroker`, as its name suggests, overrides the parent method for message broker configuration and sets:

- `setApplicationDestinationPrefixes`: Specify `/myApp` as the prefix, and any client message whose destination starts with `/myApp` will be routed to the controller's message-handling methods.
- `enableSimpleBroker`: Set the broker topic to `/chatroomTopic`. Any messages whose destinations start with `/chatroomTopic` will be routed to the message broker (that is, broadcasted to other connected clients). Since we are using an in-memory broker, we can specify any topic. If we use a dedicated broker, the destination's name would be `/topic` or `/queue`, based on the subscription model (pub/sub or point-to-point).

The overridden method `registerStompEndpoints` is used to set the endpoint and fallback options. Let's look at it closely:

- The client-side WebSocket can connect to the server's endpoint at `/broadcastMyMessage`. Since STOMP has been selected as the subprotocol, we do not need to know about the underlying message format and let STOMP handle it.
- The `.withSockJS()` method enables Spring's fallback option. This guarantees successful WebSocket communication in any type or version of browser.

As Spring MVC forwards HTTP requests to methods in controllers, the MVC extension can receive STOMP messages over WebSocket and forward them to controller methods. A Spring `Controller` class can receive client STOMP messages whose destinations start with `/myApp`. The handler method can reply to subscribed clients by sending the returned message to the broker channel, and the broker replies to the client by sending the message to the response channel. At the end of this section, we will look at some more information about the messaging architecture. As an example, let's look at the `ChatroomController` class:

```
    package com.springessentialsbook.chapter4;
      ...
@Controller
public class ChatroomController {

    @MessageMapping("/broadcastMyMessage")
    @SendTo("/chatroomTopic/broadcastClientsMessages")
```

```
    public ReturnedDataModelBean
    broadCastClientMessage(ClientInfoBean message) throws
    Exception {
        String returnedMessage=message.getClientName() +
        ":"+message.getClientMessage();
        return new ReturnedDataModelBean(returnedMessage );
    }
}
```

Here, `@Controller` tags `ChatroomController` as an MVC workflow controller. `@MessageMapping` is used to tell the controller to map the client message to the handler method (`broadCastClientMessage`). This will be done by matching a message endpoint to the destination (`/broadcastMyMessage`). The method's returned object (`ReturnedDataModelBean`) will be sent back through the broker to the subscriber's topic (`/chatroomTopic/broadcastClientsMessages`) by the `@SendTo` annotation. Any message in the topic will be broadcast to all subscribers (clients). Note that clients do not wait for the response, since they send and listen to messages to and from the topic and not the service directly.

Our domain POJOs (`ClientInfoBean` and `ReturnedDataModelBean`), detailed as follows, will provide the communication message payloads (actual message content) between the client and server:

```
package com.springessentialsbook.chapter4;
public class ClientInfoBean {
    private String clientName;
    private String clientMessage;
    public String getClientMessage() {
    return clientMessage;
  }
    public String getClientName() {
        return clientName;
    }
}


package com.springessentialsbook.chapter4;
public class ReturnedDataModelBean {

    private String returnedMessage;
    public ReturnedDataModelBean(String returnedMessage) {
        this.returnedMessage = returnedMessage; }
    public String getReturnedMessage() {
        return returnedMessage;
    }
}
```

To add some sort of security, we can add basic HTTP authentication, as follows (we are not going to explain Spring security in this chapter, but it will be detailed in the next chapter):

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests().anyRequest().authenticated();
    }
    @Autowired
    void configureGlobal(AuthenticationManagerBuilder auth) throws
    Exception {
        auth.inMemoryAuthentication()
        .withUser("user").password("password").roles("USER");
    }
}
```

The `@Configuration` tags this class as a configuration class and `@EnableGlobalMethodSecurity` and `@EnableWebSecurity` set security methods and web security in the class. In the `configure` method, we set basic authentication, and in `configureGlobal`, we set the recognized username and password as well as the role that the user belongs to.

To add Spring Security features, we should add the following Maven dependencies:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-messaging</artifactId>
    <version>4.0.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
</dependency>
```

As we explained in the previous section, the `@SpringBootApplication` tag sets up a Spring application within a web application without us having to write a single line of XML configuration (`applicationContext.xml` or `web.xml`):

```
package com.springessentialsbook.chapter4;
...
@SpringBootApplication
public class ChatroomBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(ChatroomBootApplication.class,
        args);
    }
}
```

Finally, you can run the application by running this command:

```
mvn spring-boot:run -Dserver.port=8090
```

This runs and deploys the web application on an embedded web server on port `8090` (`8080` is not used as it may conflict with your running Apache service). So, the index page of the application will be accessible at `http://localhost:8090/` (follow `read-me.txt` to run the application):

```
    <script src="sockjs-0.3.4.js"></script>
    <script src="stomp.js"></script>
    <script type="text/javascript">
...
function joinChatroom() {
    var topic='/chatroomTopic/broadcastClientsMessages';
    var servicePath='/broadcastMyMessage';
    var socket = new SockJS(servicePath);
    stompClient = Stomp.over(socket);
    stompClient.connect('user','password', function(frame) {
        setIsJoined(true);
        console.log('Joined Chatroom: ' + frame);
        stompClient.subscribe(topic, function(serverReturnedData){
            renderServerReturnedData(JSON.parse
            (serverReturnedData.body).returnedMessage);
        });
    });
}
...
function sendMyClientMessage() {
    var serviceFullPath='/myApp/broadcastMyMessage';
    var myText = document.getElementById('myText').value;
```

```
        stompClient.send(serviceFullPath, {}, JSON.stringify({
            'clientName': 'Client-'+randomnumber,
            'clientMessage':myText}));
        document.getElementById('myText').value='';
    }
```

On the client side, notice how the browser connects (with `joinChatRoom`) and sends data (in the `sendMyClientMessage` method). These methods use the JavaScript libraries SockJS and Stomp.js.

As you can see, when a client subscribes to a topic, it registers a listener method (`stompClient.subscribe(topic, function(serverReturnedData){….}`). The listener method will be called when any message (from any client) arrives in the topic.

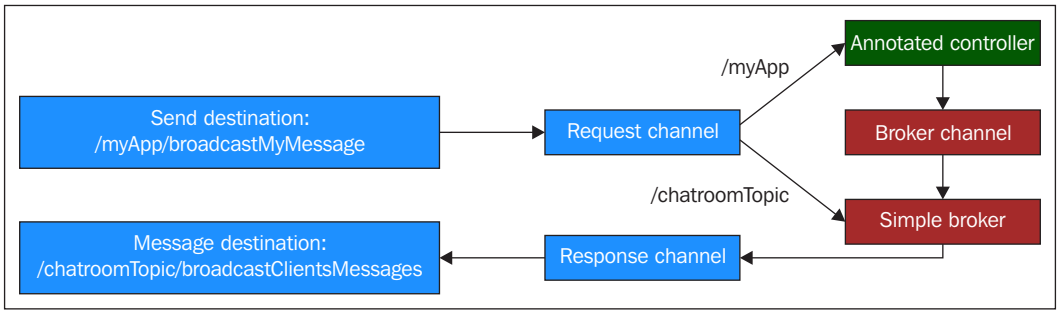
As discussed earlier, some versions of browsers do not support WebSocket. SockJS was introduced to handle all versions of browsers. On the client side, when you try to connect to the server, the SockJS client sends the `GET/info` message to get some information from the server. Then it chooses the transport protocol, which could be one of WebSocket, HTTP streaming, or HTTP long-polling. WebSocket is the preferred transport protocol; however, in case of browser incompatibility, it chooses HTTP streaming, and in the worse case, HTTP long-polling.

In the beginning of this section, we described how our WebSocket application integrates with the Spring messaging system and works in a way similar to legacy messaging systems.

The overridden method settings of `@EnableWebSocketMessageBroker` and `ChatroomWebSocketMessageBrokerConfigurer` create a concrete message flow (refer to the following diagram). In our messaging architecture, channels decouple receivers and senders. The messaging architecture contains three channels:

- The client inbound channel (**Request channel**) for request messages sent from the client side
- The client outbound channel (**Response channel**) for messages sent to the client side
- The **Broker channel** for internal server messages to the broker

Our system uses STOMP destinations for simple routing by prefix. Any client message whose destination starts with `/myApp` will be routed to controller message-handling methods. Any message whose destination starts with `/chatroomTopic` will be routed to the message broker.



The simple broker (in-memory) messaging architecture

Here is the messaging flow of our application:

1. The client connects to the WebSocket endpoint (`/broadcastMyMessage`).
2. Client messages to `/myApp/broadcastMyMessage` will be forwarded to the `ChatroomController` class (through the **Request channel**). The mapping controller's method passes the returned value to the Broker channel for the topic `/chatroomTopic/broadcastClientsMessages`.
3. The broker passes the message to the **Response channel**, which is the topic `/chatroomTopic/broadcastClientsMessages`, and clients subscribed to this topic receive the message.

Broadcasting a message to a single user in a WebSocket application

In the previous section, we saw a WebSocket application of the multiple subscriber model, in which a broker sent messages to a topic. Since all clients had subscribed to the same topic, all of them received messages. Now, you are asked to develop an application that targets a specific user in a WebSocket chat application.

Suppose you want to develop an automated answering application in which a user sends a question to the system and gets an answer automatically. The application is almost the same as the previous one (STOMP over WebSocket and the fallback option in Spring 4), except that we should change the WebSocket configurer and endpoint on the server side and subscription on the client side. The code for the `AutoAnsweringWebSocketMessageBrokerConfigurer` class is:

```
@Configuration
@EnableWebSocketMessageBroker
public class AutoAnsweringWebSocketMessageBrokerConfigurer extends
AbstractWebSocketMessageBrokerConfigurer {
   @Override
   public void configureMessageBroker(MessageBrokerRegistry
   config) {
      config.setApplicationDestinationPrefixes("/app");
      config.enableSimpleBroker("/queue");
      config.setUserDestinationPrefix("/user");
   }
   @Override
   public void registerStompEndpoints(StompEndpointRegistry
   registry) {
      registry.addEndpoint("/message").withSockJS();
   }
}
```

The `config.setUserDestinationPrefix("/user")` method sets a prefix noting that a user has subscribed and expects to get their own message on the topic.
The code for the `AutoAnsweringController` class is:

```
@Controller
public class AutoAnsweringController {
    @Autowired
    AutoAnsweringService autoAnsweringService;
    @MessageMapping("/message")
    @SendToUser
    public String sendMessage(ClientInfoBean message) {
        return autoAnsweringService.answer(message);
    }
    @MessageExceptionHandler
    @SendToUser(value = "/queue/errors", broadcast = false)
    String handleException(Exception e) {
        return "caught ${e.message}";
    }
```

```
}


@Service
public class AutoAnsweringServiceImpl implements AutoAnsweringService
{
    @Override
    public String answer(ClientInfoBean bean) {
        StringBuilder mockBuffer=new StringBuilder();
        mockBuffer.append(bean.getClientName())
                .append(", we have received the message:")
                .append(bean.getClientMessage());
        return mockBuffer.toString();
    }
}
```

In the endpoint, we use `@SendToUser` instead of `@SendTo("...")`. This forwards the response only to the sender of the message. `@MessageExceptionHandler` will send errors `(broadcast = false)` to the sender of message as well.

`AutoAnsweringService` is just a mock service to return an answer to the client message. On the client side, we only add the `/user` prefix when a user subscribes to the topic (`/user/queue/message`):

```
function connectService() {
    var servicePath='/message';
    var socket = new SockJS(servicePath);
    stompClient = Stomp.over(socket);
    stompClient.connect({}, function(frame) {

        setIsJoined(true);
        stompClient.subscribe('/user/queue/message',
        function(message) {
            renderServerReturnedData(message.body);
        });
        stompClient.subscribe('/user/queue/error',
        function(message) {
            renderReturnedError(message.body);
        });
    });
}
function sendMyClientMessage() {
    var serviceFullPath='/app/message';
```

```
        var myText = document.getElementById('myText').value;
        stompClient.send(serviceFullPath, {}, JSON.stringify({
        'clientName': 'Client-'+randomnumber,
        'clientMessage':myText}));
        document.getElementById('myText').value='';
    }
```

The topic `user/queue/error` is used to receive errors dispatched from the server side.

For more about Spring's WebSocket support, go to `http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/websocket.html`. For more about WebSocket communication, refer to *Chapter 8*, *Replacing HTTP with WebSockets* from the book *Enterprise Web Development*, *Yakov Fain, Victor Rasputnis, Anatole Tartakovsky, Viktor Gamov*, *O'Reilly*.

Summary

In this chapter, we explained WebSocket-based communication, how Spring 4 has been upgraded to support WebSocket, and the fallback option to overcome browsers' WebSocket incompatibility. We also had a small sample of adding basic HTTP authentication, which is a part of Spring Security. We will discuss more on security in *Chapter 5*, *Securing Your Applications*.

Where to buy this book

You can buy Spring Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.